CHAPTER 4

The MongoDB Data Model

> *"MongoDB is designed to work with documents without any need of predefined columns or data types (unlike relational databases), making the data model extremely flexible."*

In this chapter, you will learn about the MongoDB data model. You will also learn what flexible schema (polymorphic schema) means and why it's a significant contemplation of MongoDB data model.

The Data Model

In the previous chapter, you saw that MongoDB is a document-based database system where the documents can have a flexible schema. This means that documents within a collection can have different (or same) sets of fields. This affords you more flexibility when dealing with data.

In this chapter, you will explore MongoDB's flexible data model. Wherever required, we will demonstrate the difference in the approach compared to RDBMS systems.

A MongoDB deployment can have many databases. Each database is a set of collections. Collections are similar to the concept of tables in SQL; however, they are schemaless. Each collection can have multiple documents. Think of a document as a row in SQL. Figure 4-1 depicts the MongoDB database model.

S.G. Edward and N. Sabharwal, *Practical MongoDB*, DOI 10.1007/978-1-4842-0647-8_4

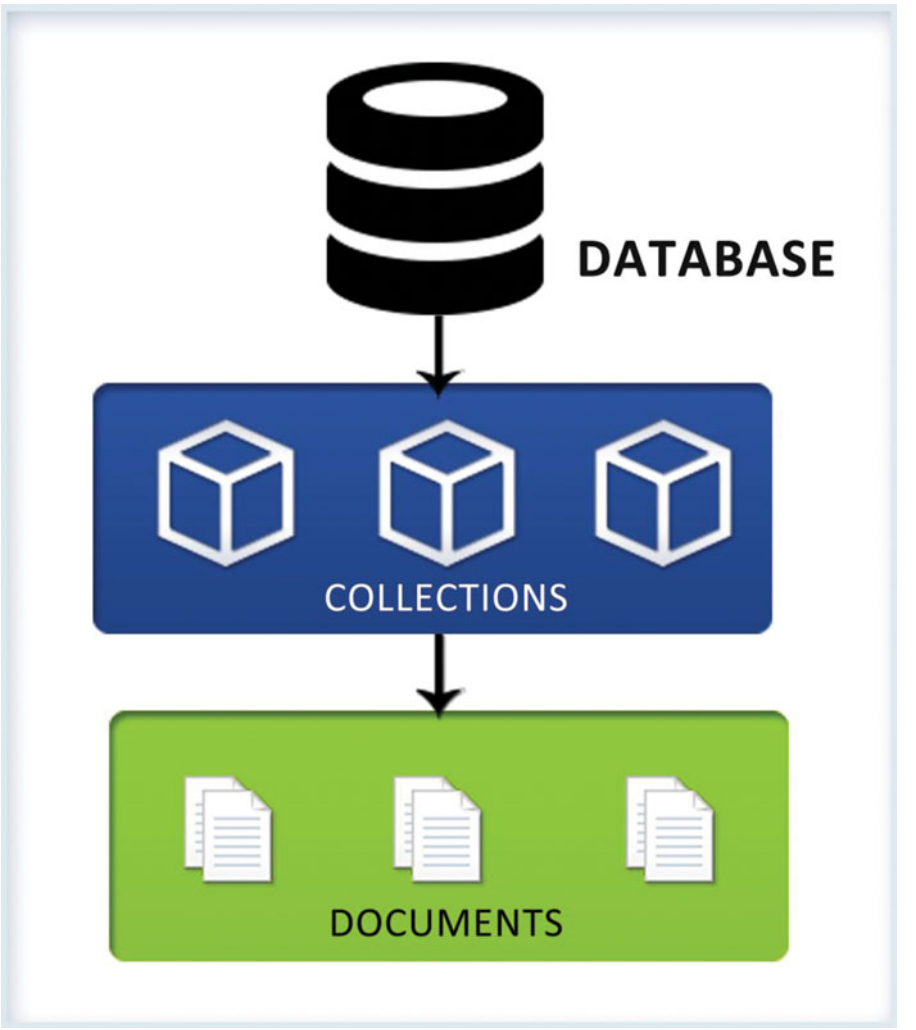

Figure 4-1. *MongoDB database model*

In an RDBMS system, since the table structures and the data types for each column are fixed, you can only add data of a particular data type in a column. In MongoDB, a collection is a collection of documents where data is stored as key-value pairs.

Let's understand with an example how data is stored in a document. The following document holds the name and phone numbers of the users:

```
{"Name": "ABC", "Phone": ["1111111",      "222222" ] }
```

Dynamic schema means that documents within the same collection can have the same or different sets of fields or structure, and even common fields can store different types of values across documents. There's no rigidness in the way data is stored in the documents of a collection.

Let's see an example of a Region collection:

```
{ "R_ID" : "REG001",  "Name" : "United States" }
{ "R_ID" :1234,  "Name" : "New York" , "Country" : "United States" }
```

In this code, you have two documents in the Region collection. Although both documents are part of a single collection, they have different structures: the second collection has an additional field of information, which is country. In fact, if you look at the "R_ID" field, it stores a STRING value in the first document whereas it's a number in the second document.

Thus a collection's documents can have entirely different schemas. It falls to the application to store the documents in a particular collection together or to have multiple collections.

JSON and BSON

MongoDB is a document-based database. It uses Binary JSON for storing its data.

In this section, you will learn about JSON and Binary-JSON (BSON). JSON stands for JavaScript Object Notation. It's a standard used for data interchange in today's modern Web (along with XML). The format is human and machine readable. It is not only a great way to exchange data but also a nice way to store data.

All the basic data types (such as strings, numbers, Boolean values, and arrays) are supported by JSON.

The following code shows what a JSON document looks like:

```
{
"_id" : 1,
"name" : { "first" : "John", "last" : "Doe" },
"publications" : [
           {
             "title" : "First Book",
             "year" : 1989,
             "publisher" : "publisher1"
           },
           { "title" : "Second Book",
             "year" : 1999,
             "publisher" : "publisher2"
           }
]
}
```

JSON lets you keep all the related pieces of information together in one place, which provides excellent performance. It also enables the updating of a document to be independent. It is schemaless.

Binary JSON (BSON)

MongoDB stores the JSON document in a binary-encoded format. This is termed as BSON. The BSON data model is an extended form of the JSON data model.

MongoDB's implementation of a BSON document is fast, highly traversable, and lightweight. It supports embedding of arrays and objects within other arrays, and also enables MongoDB to reach inside the objects to build indexes and match objects against queried expressions, both on top-level and nested BSON keys.

The Identifier (_id)

You have seen that MongoDB stores data in documents. Documents are made up of key-value pairs. Although a document can be compared to a row in RDBMS, unlike a row, documents have flexible schema. A key, which is nothing but a label, can be roughly compared to the column name in RDBMS. A key is used for querying data from the documents. Hence, like a RDBMS primary key (used to uniquely identify each row), you need to have a key that uniquely identifies each document within a collection. This is referred to as _id in MongoDB.

If you have not explicitly specified any value for a key, a unique value will be automatically generated and assigned to it by MongoDB. This key value is immutable and can be of any data type except arrays.

Capped Collection

You are now well versed with collections and documents. Let's talk about a special type of collection called a capped collection.

MongoDB has a concept of capping the collection. This means it stores the documents in the collection in the inserted order. As the collection reaches its limit, the documents will be removed from the collection in FIFO (first in, first out) order. This means that the least recently inserted documents will be removed first.

This is good for use cases where the order of insertion is required to be maintained automatically, and deletion of records after a fixed size is required. One such use cases is log files that get automatically truncated after a certain size.

■ **Note** MongoDB itself uses capped collections for maintaining its replication logs. Capped collection guarantees preservation of the data in insertion order, so queries retrieving data in the insertion order return results quickly and don't need an index. Updates that change the document size are not allowed.

Polymorphic Schemas

As you are already conversant with the schemaless nature of MongoDB data structure, let's now explore polymorphic schemas and use cases.

A polymorphic schema is a schema where a collection has documents of different types or schemas. A good example of this schema is a collection named Users. Some user documents might have an extra fax number or email address, while others might have only phone numbers, yet all these documents coexist within the same Users collection. This schema is generally referred to as a polymorphic schema.

In this part of the chapter, you'll explore the various reasons for using a polymorphic schema.

Object-Oriented Programming

Object-oriented programming enables you to have classes share data and behaviors using inheritance. It also lets you define functions in the parent class that can be overridden in the child class and thus will function differently in a different context. In other words, you can use the same function name to manipulate the child as well as the parent class, although under the hood the implementations might be different. This feature is referred to as polymorphism.

The requirement in this case is the ability to have a schema wherein all of the related sets of objects or objects within a hierarchy can fit in together and can also be retrieved identically.

Let's consider an example. Suppose you have an application that lets the user upload and share different content types such as HTML pages, documents, images, videos, etc. Although many of the fields are common across all of the above-mentioned content types (such as Name, ID, Author, Upload Date, and Time), not all fields are identical. For example, in the case of images, you have a binary field that holds the image content, whereas an HTML page has a large text field to hold the HTML content.

In this scenario, the MongoDB polymorphic schema can be used wherein all of the content node types are stored in the same collection, such as LoadContent, and each document has relevant fields only.

```
// "Document collections" - "HTMLPage" document
{
_id: 1,
title: "Hello",
type: "HTMLpage",
text: "<html>Hi..Welcome to my world</html>"
}
...
// Document collection also has a "Picture" document
{
_id: 3,
title: "Family Photo",
type: "JPEG",
sizeInMB: 10,........
}
```

This schema not only enables you to store related data with different structures together in a same collection, it also simplifies the querying. The same collection can be used to perform queries on common fields such as fetching all content uploaded on a particular date and time as well as queries on specific fields such as finding images with a size greater than X MB.

Thus object-oriented programming is one of the use cases where having a polymorphic schema makes sense.

Schema Evolution

When you are working with databases, one of the most important considerations that you need to account for is the schema evolution (i.e. the change in the schema's impact on the running application). The design should be done in a way as to have minimal or no impact on the application, meaning no or minimal downtime, no or very minimal code changes, etc.

Typically, schema evolution happens by executing a migration script that upgrades the database schema from the old version to the new one. If the database is not in production, the script can be simple drop and recreation of the database. However, if the database is in a production environment and contains live data, the migration script will be complex because the data will need to be preserved. The script should take this into consideration. Although MongoDB offers an Update option that can be used to update all the documents' structure within a collection if there's a new addition of a field, imagine the impact of doing this if you have thousands of documents in the collection. It would be very slow and would have a negative impact on the underlying application's performance. One of the ways of doing this is to include the new structure to the new documents being added to the collection and then gradually migrating the collection in the background while the application is still running. This is one of the many use cases where having a polymorphic schema will be advantageous.

For example, say you are working with a Tickets collection where you have documents with ticket details, like so:

```
// "Ticket1" document (stored in "Tickets" collection")
{
_id: 1,
Priority: "High",
type: "Incident",
text: "Printer not working"
}..........
```

At some point, the application team decides to introduce a "short description" field in the ticket document structure, so the best alternative is to introduce this new field in the new ticket documents. Within the application, you embed a piece of code that will handle retrieving both "old style" documents (without a short description field) and "new style" documents (with a short description field). Gradually the old style documents can be migrated to the new style documents. Once the migration is completed, if required the code can be updated to remove the piece of code that was embedded to handle the missing field.

Summary

In this chapter, you learned about the MongoDB data model. You also looked at identifiers and capped collections. You concluded the chapter with an understanding of how the flexible schema helps.

In the following chapter, you will get started with MongoDB. You will perform the installation and configuration of MongoDB.